PIT: Optimization of Dynamic Sparse Deep Learning Models via Permutation Invariant Transformation

Ningxin Zheng*, Huiqiang Jiang*, Quanlu Zhang, Zhenhua Han, Lingxiao Ma,
Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, Lidong Zhou

Microsoft Research

SOSP' 23

Presented by Jiaan Zhu, Long Zhao and Qinghe Wang

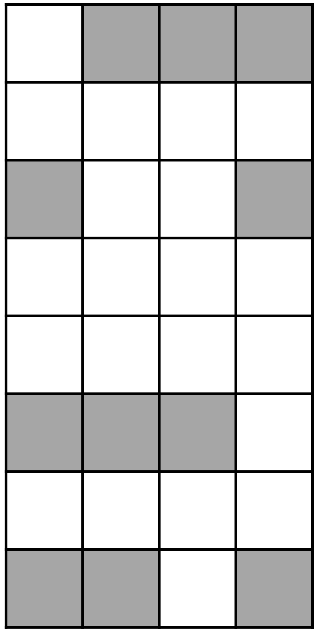
USTC, CHINA
ADSLAB
先进数据系统实验室

Outline

- Background & Challenges
- Design & Implementation
- Evaluation

Background

- Sparse
  - Tensors with many zeros (token, weight, activation, etc.)



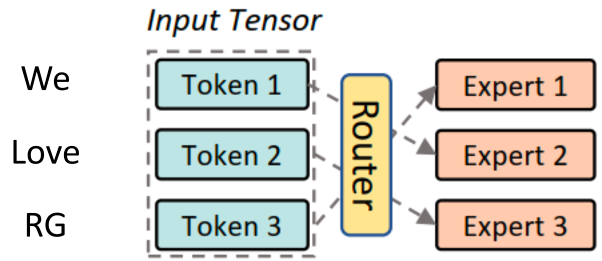
Tensor

Background

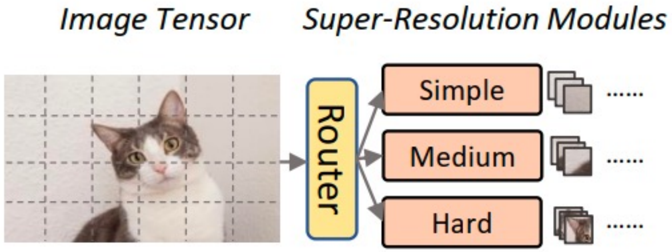
- Sparse
- Dynamic Sparse
  - Depend on inputs and is only known at runtime

Background

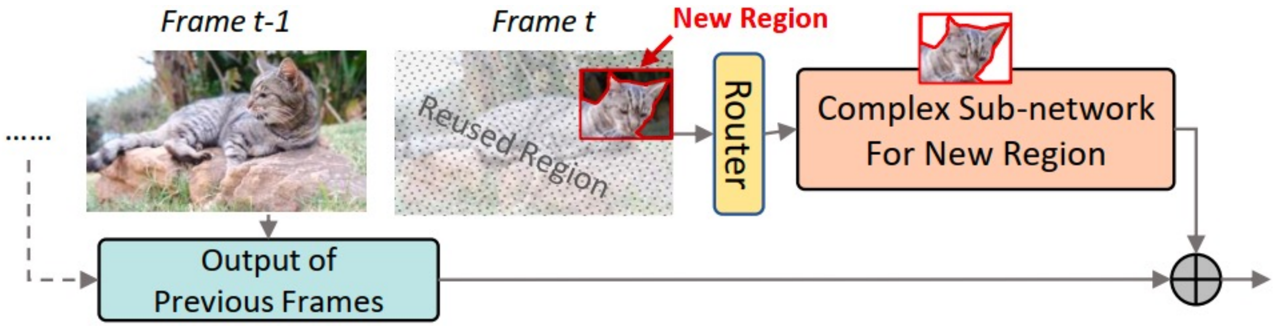
- Sparse
- Dynamic Sparse
  - ◆ Depend on inputs and is only known at runtime
  - ◆ **App-level**



(a) MoE



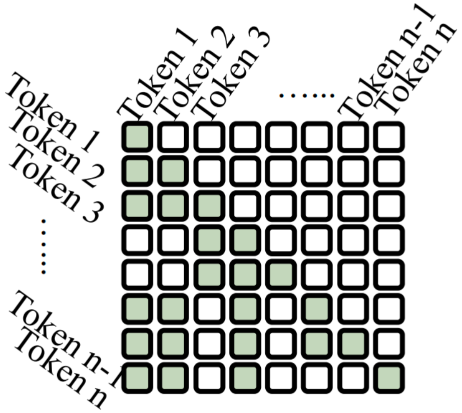
(b) Super-Resolution



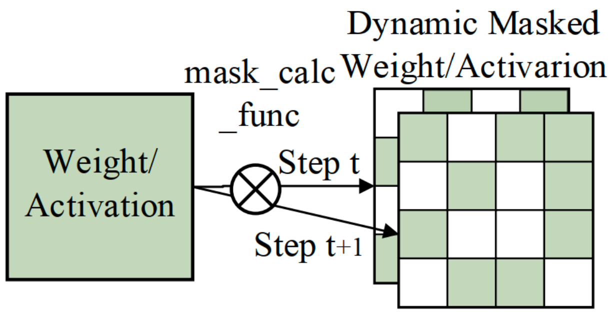
(c) Video Generation

Background

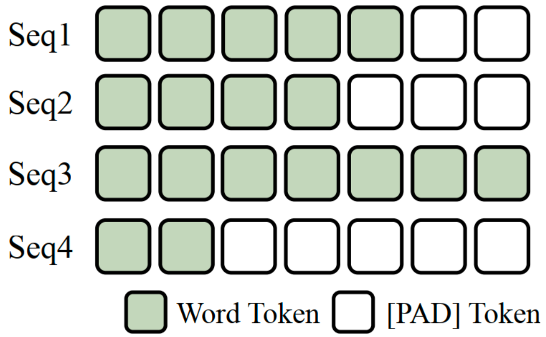
- Sparse
- Dynamic Sparse
  - ◆ Depend on inputs and is only known at runtime
  - ◆ App-level, **Tensor-level**



(a) Dynamic Attention[1,2]



(b) Sparse Training[3]

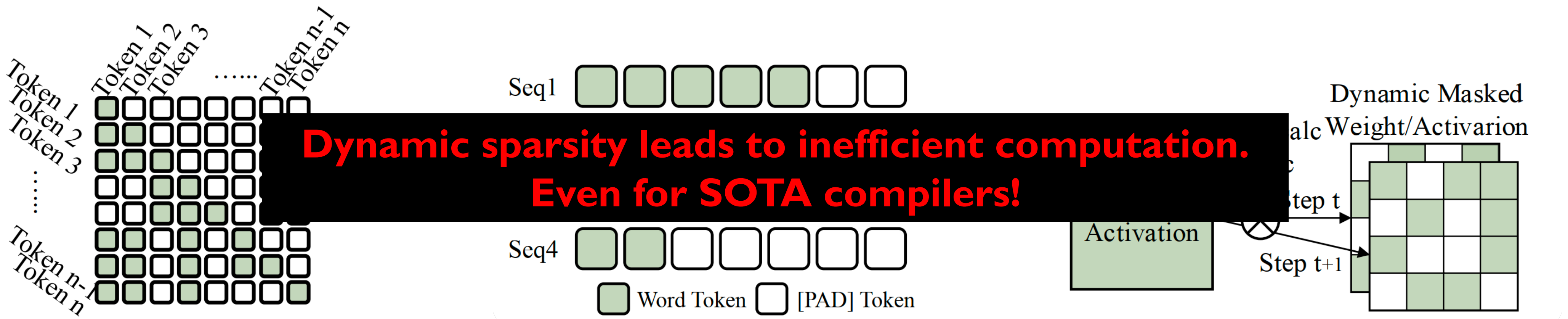


(c) Dynamic Sequence Length[4]

(1) *Generating long sequences with sparse transformers. arXiv preprint arXiv:1904.10509, 2019.*
(2) *Transformer acceleration with dynamic sparse attention. ArXiv preprint,abs/2110.11299, 2021.*
(3) *Block pruning for faster transformers. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing.*
(4) *Megablocks: Efficient sparse training with mixture-of-experts. MLSys2023, 2023.*

Background

- Sparse
- Dynamic Sparse
  - Depend on inputs and is only known at runtime
  - App-level, **Tensor-level**



Dynamic sparsity leads to inefficient computation.
Even for SOTA compilers!

(a) Dynamic Attention[1,2] (b) Dynamic Sequence Length[3] (c) Sparse Training[4]

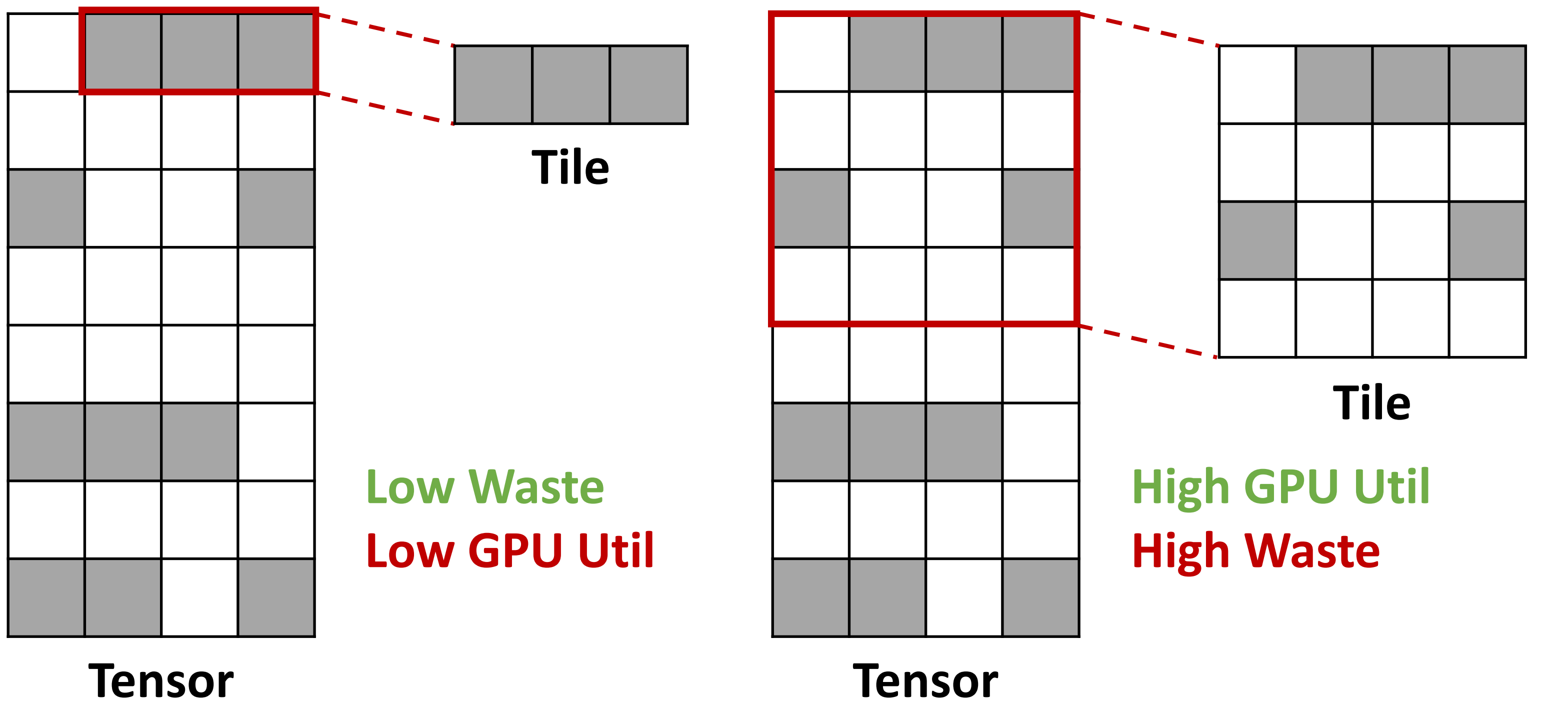
(1) *Generating long sequences with sparse transformers. arXiv preprint arXiv:1904.10509, 2019.*
(2) *Transformer acceleration with dynamic sparse attention. ArXiv preprint,abs/2110.11299, 2021.*
(3) *Megablocks: Efficient sparse training with mixture-of-experts. MLSys2023, 2023.*
(4) *Block pruning for faster transformers. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing.*

Background

- Tiling
  - Split tensor into smaller slices (tiles)
  - Reusing cached tiles can reduces the amount of data movement
  - Choosing an appropriate size can optimize data reuse

Background

- Tiling
- Tiling with Dynamic Sparsity
  - Trade-off exists between efficient tiling & sparsity shape alignment



Existing Solutions

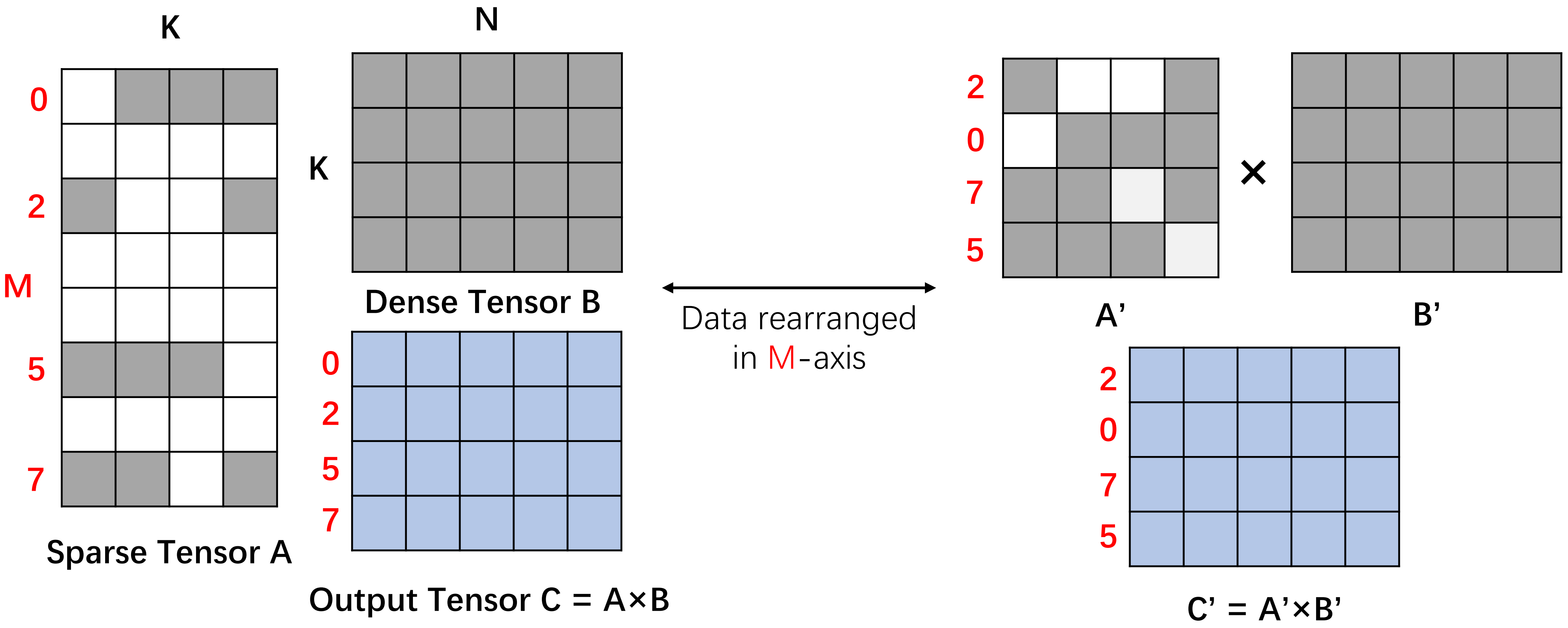
	Compiler/Library	Sparsity Aware	Dynamic Sparsity	Low Overhead
	Triton	☹	☹	☺
	ROLLER [OSDI’22]	☹	☹	☺
Specialized GPU Kernels	TVM-sparsity	☺	☹	☹
Specialized GPU Kernels	SparTA [OSDI’22]	☺	☹	☹
Convert to Special Format	cuSparse	☺	☺	☹
Convert to Special Format	Spunik [SC’20]	☺	☺	😐
	PIT [SOSP’23]	☺	☺	☺

Goal

- Try to find the most efficient tiling scheme
  - Minimize zero values
  - Maximize parallelism
  - Minimize latency

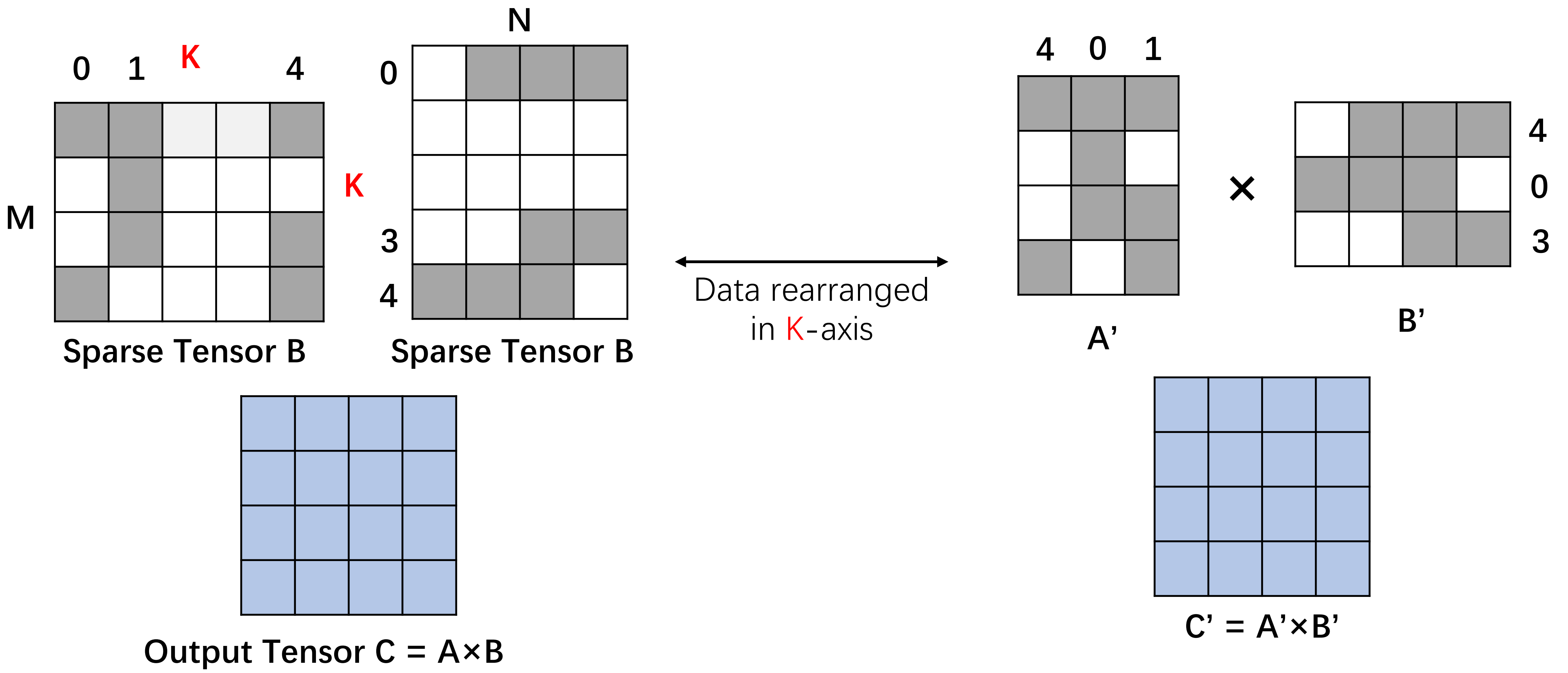
Opportunity

- Sparse data can be merge to efficient dense tile
  - With equivalent computation



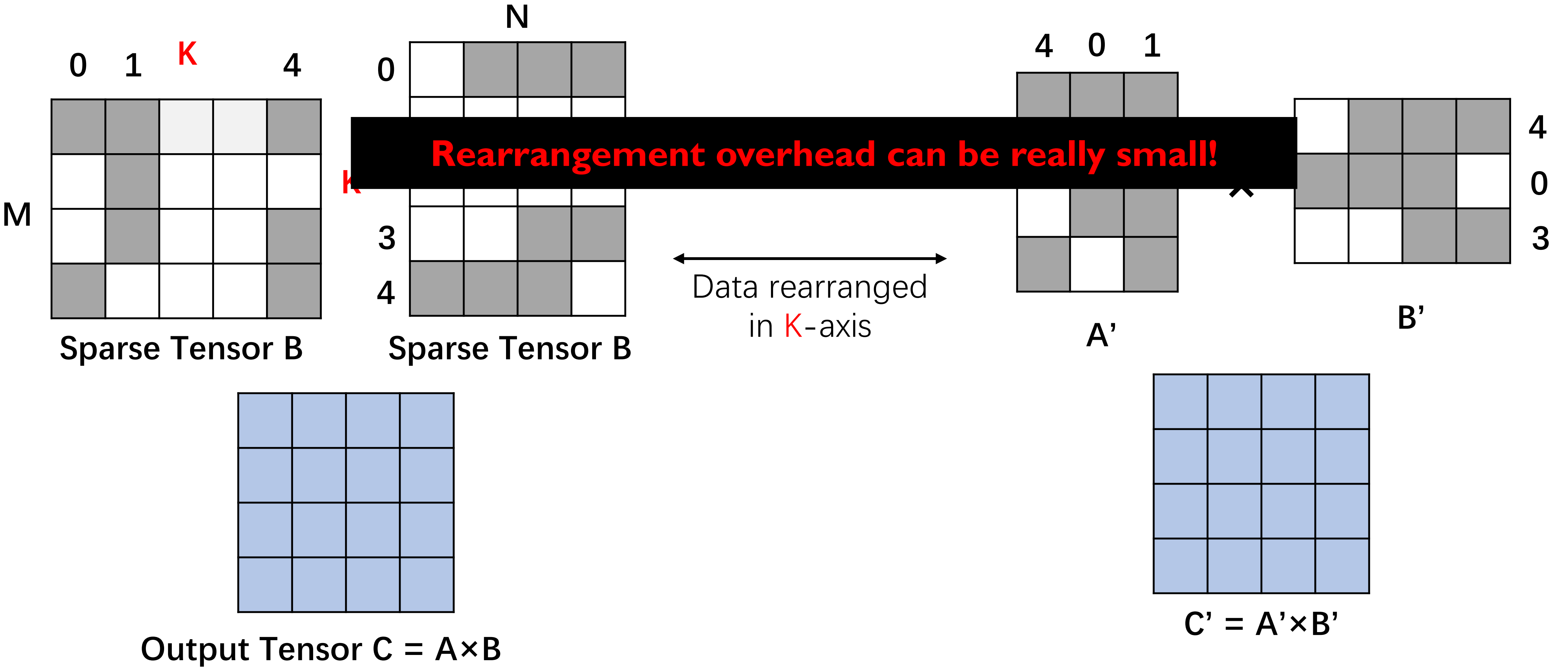
Opportunity

- Sparse data can be merge to efficient dense tile
  - With equivalent computation

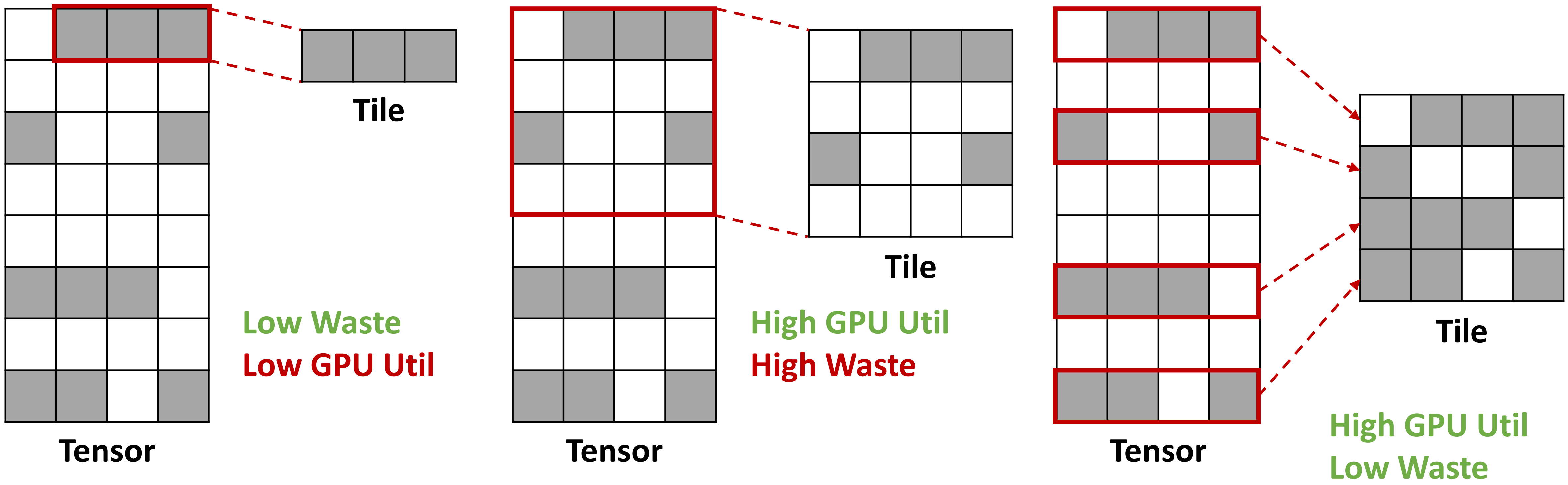


Opportunity

- Sparse data can be merge to efficient dense tile
  - ◆ With equivalent computation



Idea


Tile
Low Waste
Low GPU Util
Tensor
Tile
High GPU Util
High Waste
Tensor
Tile
Tensor
High GPU Util
Low Waste

Outline

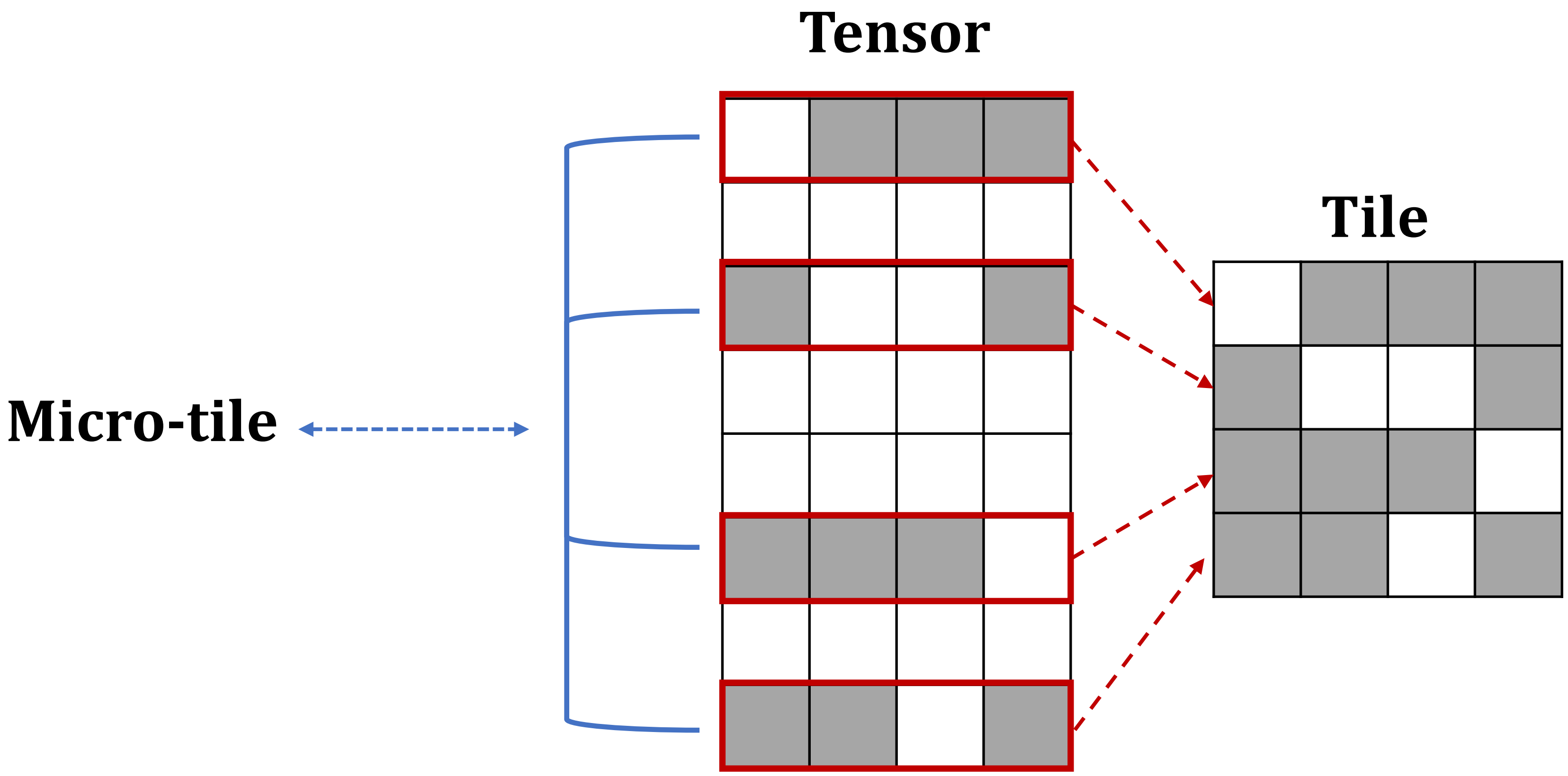
- Background & Challenges
- Design & Implementation
- Evaluation

PIT Overview

- New Transformation Mechanism of PIT
- How to select Micro-tile and Kernel configuration
- Conversion Optimization for lower overhead

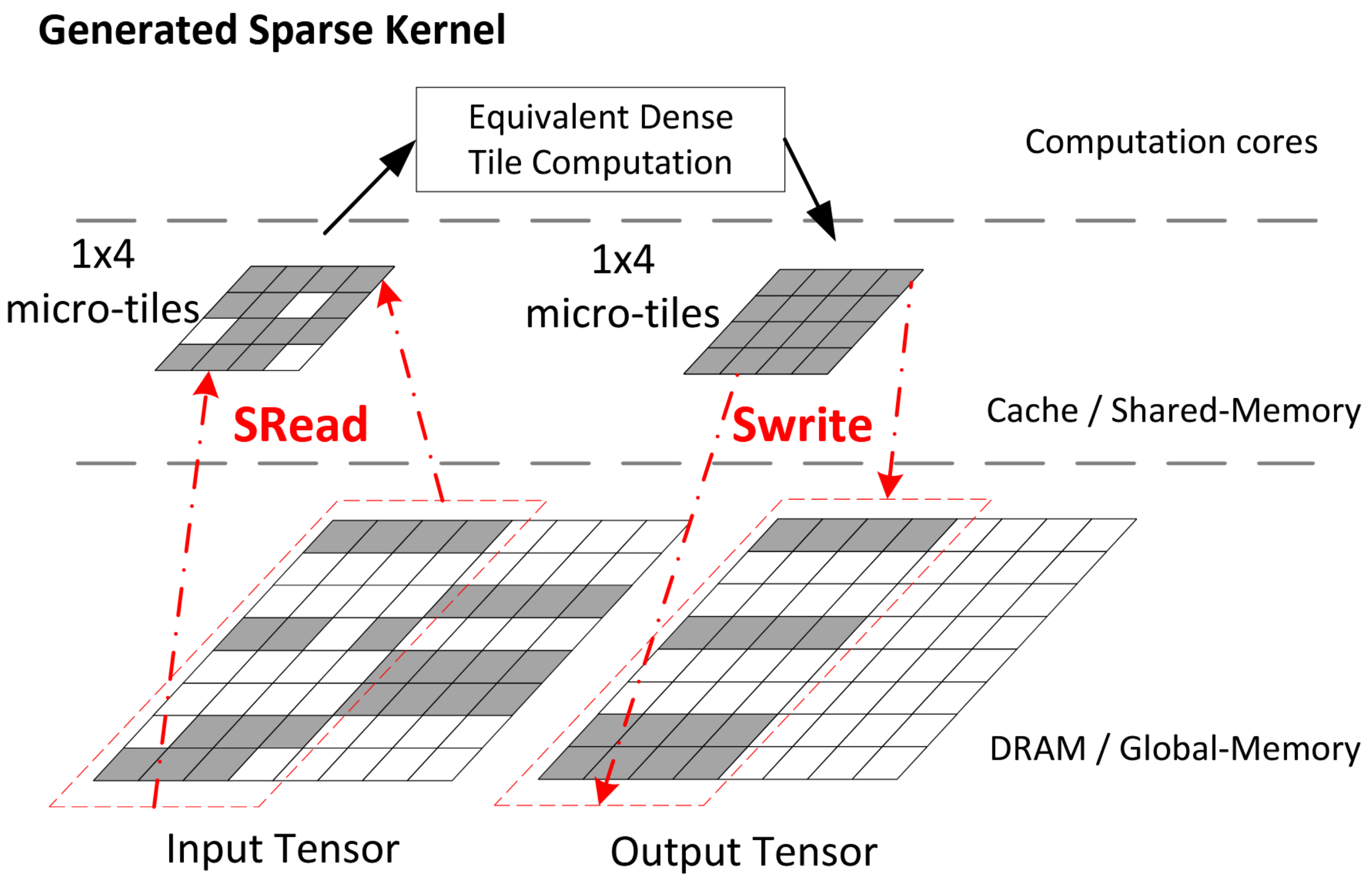
PIT Transformation Mechanism

- Micro-tile

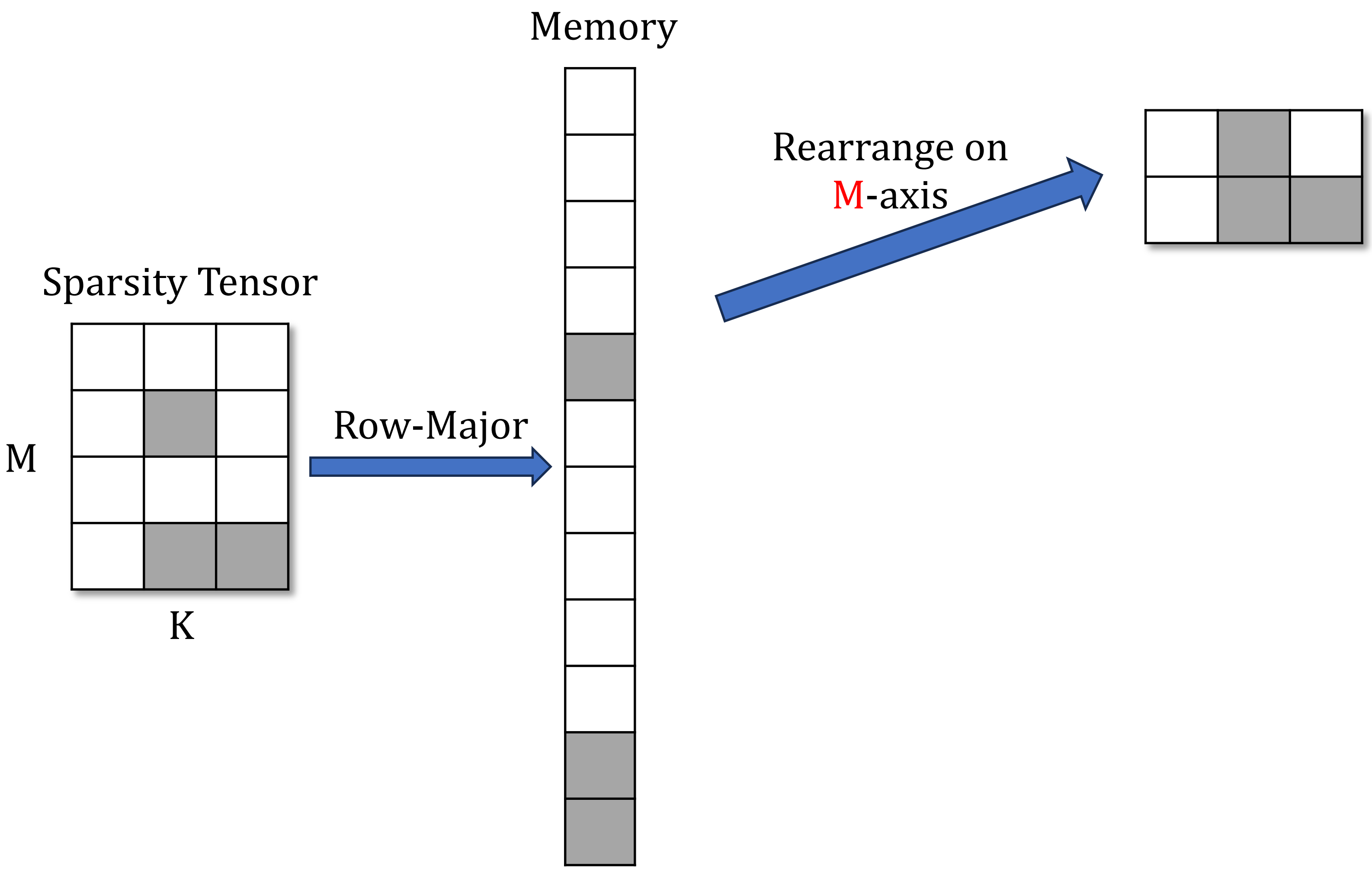


PIT Transformation Mechanism

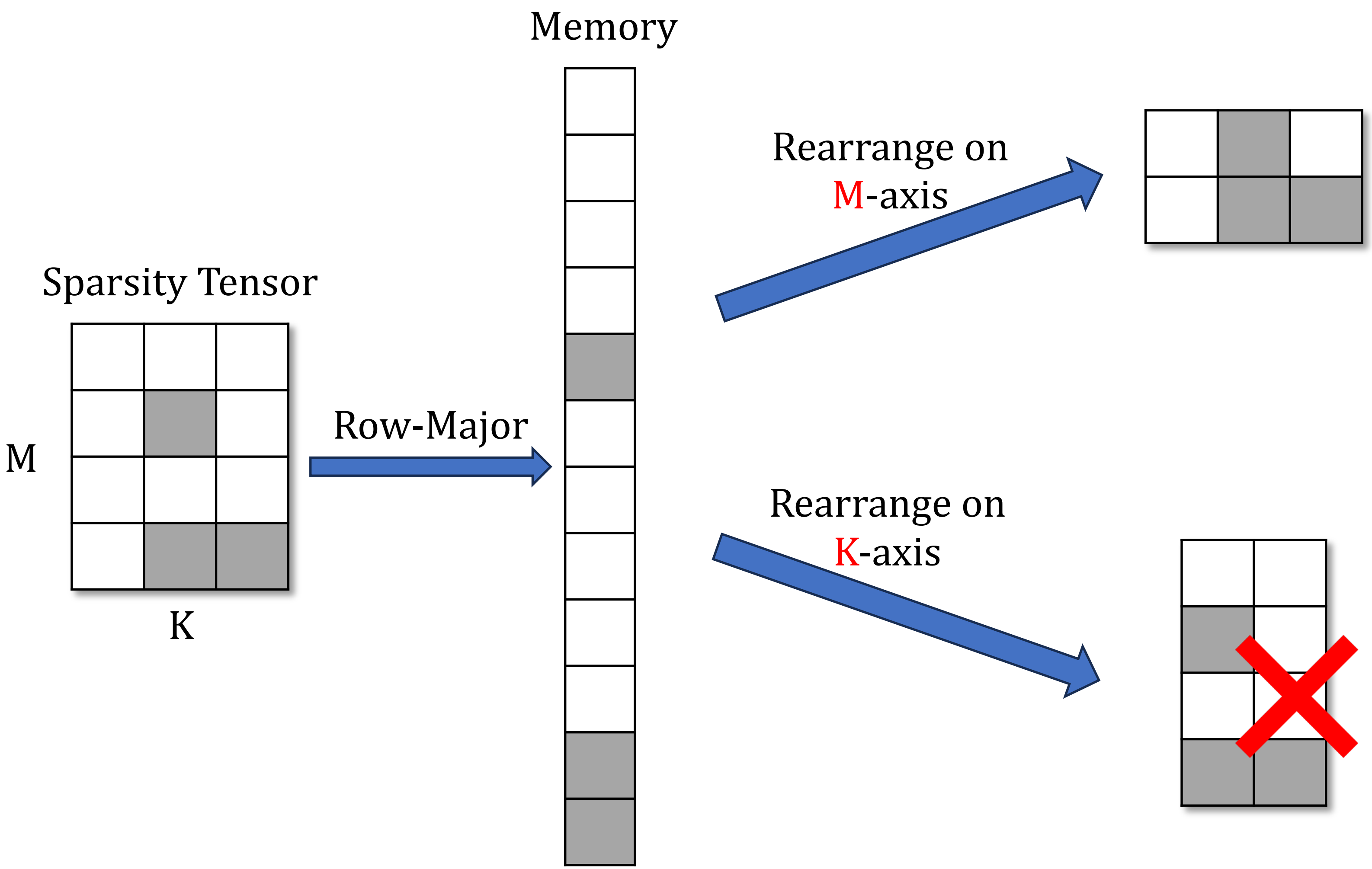
- New Primitives: SRead and SWrite



Micro-tile and Kernel Selection



Micro-tile and Kernel Selection



Micro-tile and Kernel Selection

Algorithm 1: Kernel selection for a dynamic sparsity operator.

Data: Op: A dynamically sparse operator,
D_{sparse}: A list of n sparsity samples of Op.
Result: $Best$: The best computation tile for Op.

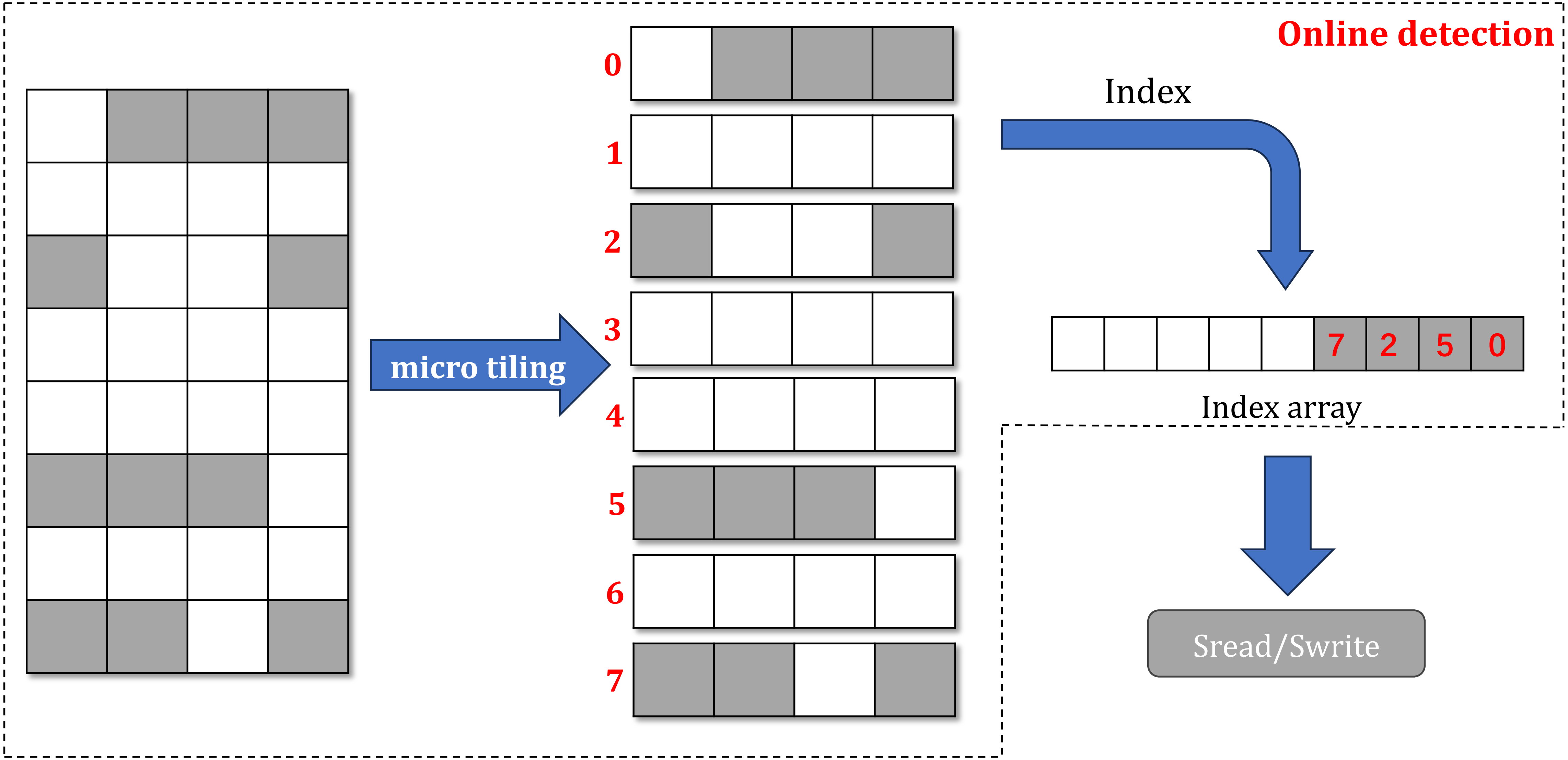
Function *KernelSelection*(D_{sparse}, Op):
  $Best$ = null; $Cost_{optimal}$ = inf;
  foreach $T \in$ *GetTilesFromTileDB*(Op) **do**
    foreach $A \in$ *GetPITAxis*(Op) **do**
      $Cost = 0$;
      $micro\_tile = GetMicroTile(T.SparseTensor, A)$;
      foreach $D \in D_{sparse}$ **do**
        Num_{tiles} = CoverAlgo(D, $micro\_tile$, A);
        $Cost \mathrel{+}= Num_{tiles} * T.tile\_cost$;
      if $Cost < Cost_{optimal}$ **then**
        $Best = S$;
        $Cost_{optimal} = Cost$;
  return $Best$;

Related to hardware instructions

Related to hardware instructions

Cost model

Online Sparsity Detection



Outline

- Background & Challenges
- Design & Implementation
- Evaluation

Evaluation

- Setup
- End-to-End Inference
  - Latency
  - Memory
- End-to-End Training
  - Latency
  - Memory
- Effectiveness of PIT Transformation
- Conversion Overhead
- Micro-Tile Online Searching

Evaluation: Concerns

- Questions to answer:
  - Q1: PIT’s advantages compared to baselines?
  - Q2: PIT’s performance in inference and training?
  - Q3: PIT’s conversion overhead?
  - Q4: Why certain baseline outperform other baselines?

Evaluation: Setup

- Baselines
  - PyTorch v1.11.0: Deep learning framework
  - DeepSpeed: Inference frameworks
    - Features: **Fuse** a layer into one operator in inference(but not in training)
  - TurboTransformers [SIGPLAN'21]:Inference frameworks
    - Features: optimized the **memory management** for varying input length
  - Tutel: Specific optimization techniques
    - Features: A efficient Mixture of Experts (MoE) implementation library

Evaluation: Setup

- Baselines
  - MegaBlocks [ML-Sys'23]: Inference frameworks
    - Features: Identify **blocky zero-element regions**, and skip them during calculations
  - SparTA [OSDI'22]: An optimization framework for **static** sparsity.
    - Features: Use efficient kernel calculations based on different sparse patterns
  - PyTorch-S: A variant of PyTorch that uses backends: cuSPARSE, Sputnik, Triton
    - cuSPARSE: mainly use Compressed Sparse Row (CSR) , provides efficient computational kernels
    - Sputni [SC'20]: Analyze the sparse pattern of the input matrix and select the most appropriate storage format and algorithm
    - Triton: A compiler and programming language, designed to simplify the process of writing GPU cores with high-performance

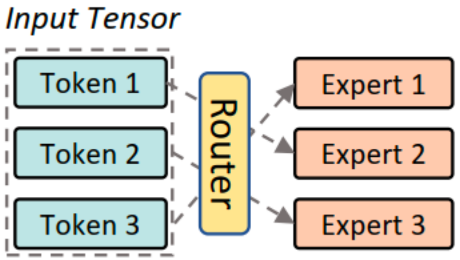
Evaluation: Setup

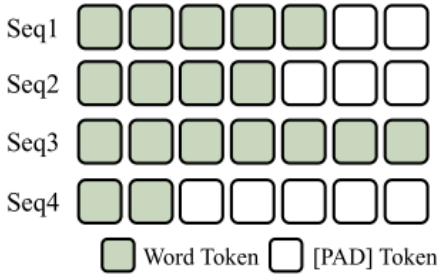
- Datasets and Hardware setup:

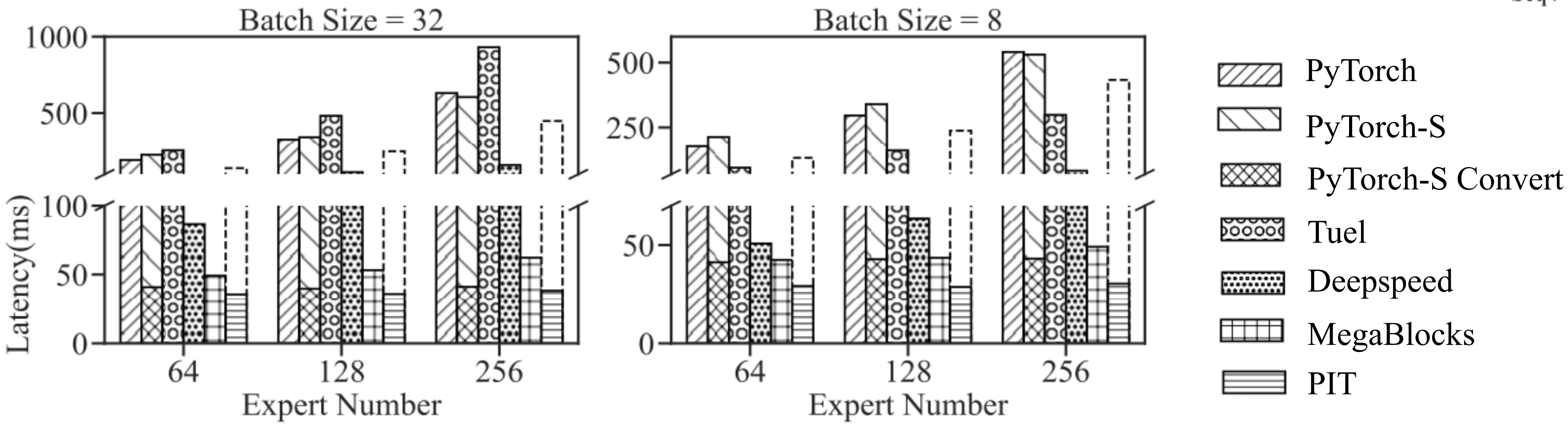
Models	Datasets	Model Structure	Precision	Devices
Switch Transformers[29]	MNLI [59]	Encoder Decoder MoE	fp16,fp32	A100
Swin-MoE [37]	ImageNet	Encoder MoE	fp16	A100
OPT [66]	Alpaca [58]	Decoder	fp32	V100
BERT [22]	GLUE [59], News [27] etc.	Encoder	fp32	V100
Longformer [14]	Arxiv [21]	Encoder	fp32	V100
MuseFormer [65]	LMD [54]	Decoder	fp32	V100

Evaluation: End-to-End Inference

- Switch Transformer (1x A100, FP16/FP32)
  - ◆ Latency: MegaBlocks stands out, but PIT is better
    - ▪ Without padding overheads
    - ▪ Simultaneous execution in MoE layers
    - ▪ Low data reorganization cost



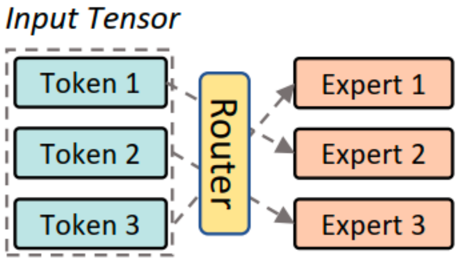


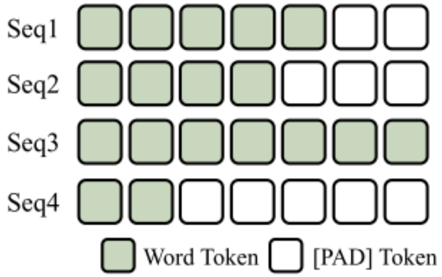


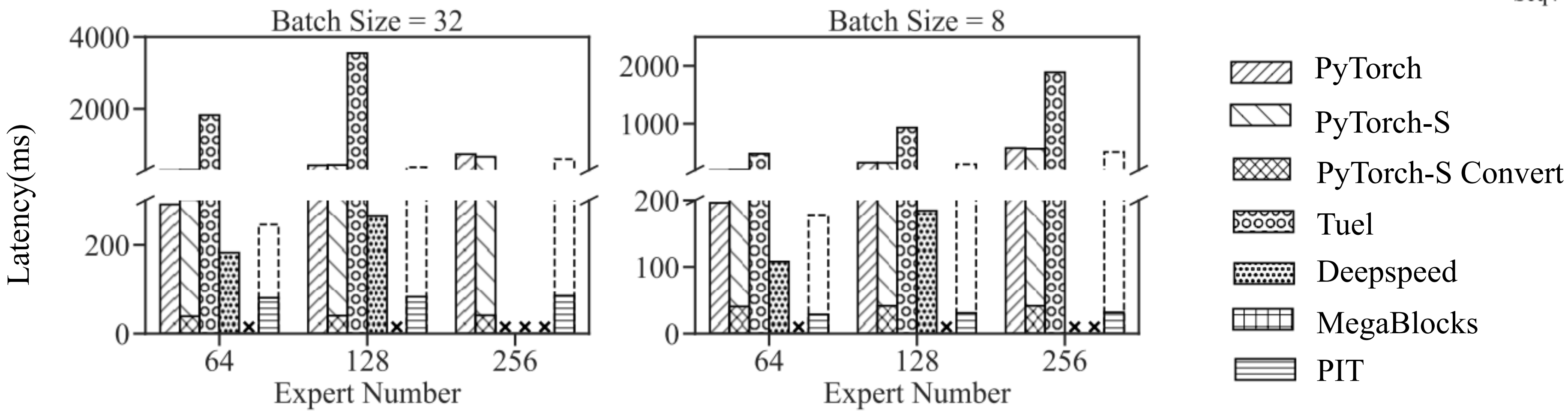
FP16 Latency

Evaluation: End-to-End Inference

- Switch Transformer (1x A100, FP16/FP32)
  - ◆ Latency: PIT is the lowest in FP32
    - Without padding overheads
    - Simultaneous execution in MoE layers
    - Low data reorganization cost



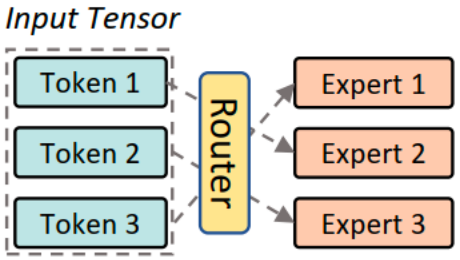


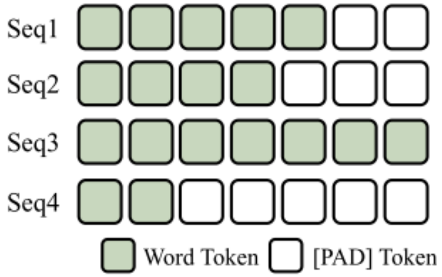


FP32 Latency

Evaluation: End-to-End Inference

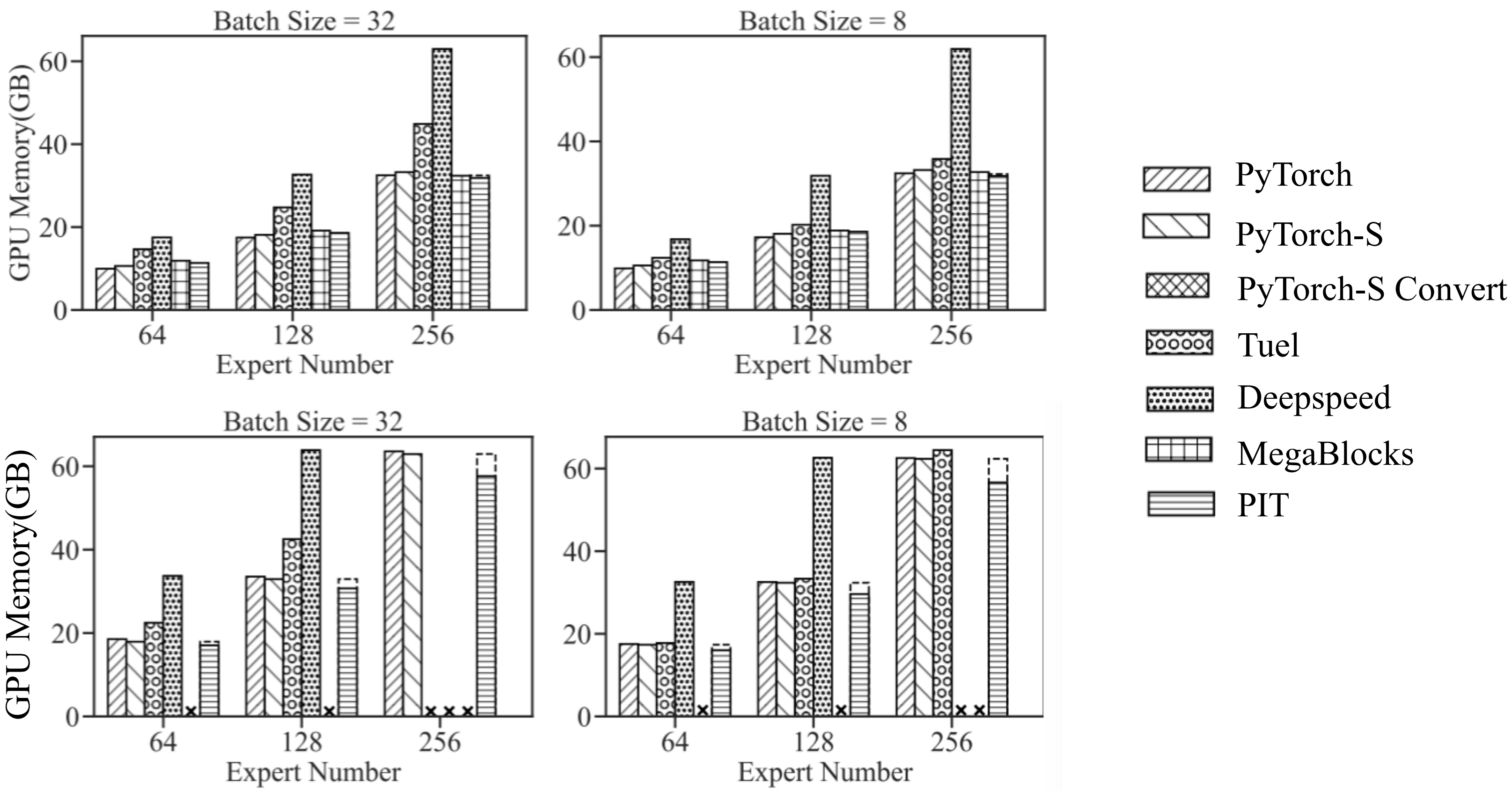
- Switch Transformer (1x A100, FP16/FP32)
  - Memory: PIT is the lowest in FP16 and FP32
    - Without padding





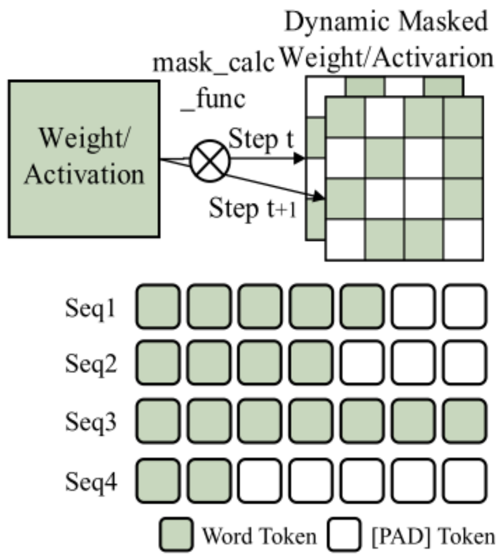
FP16

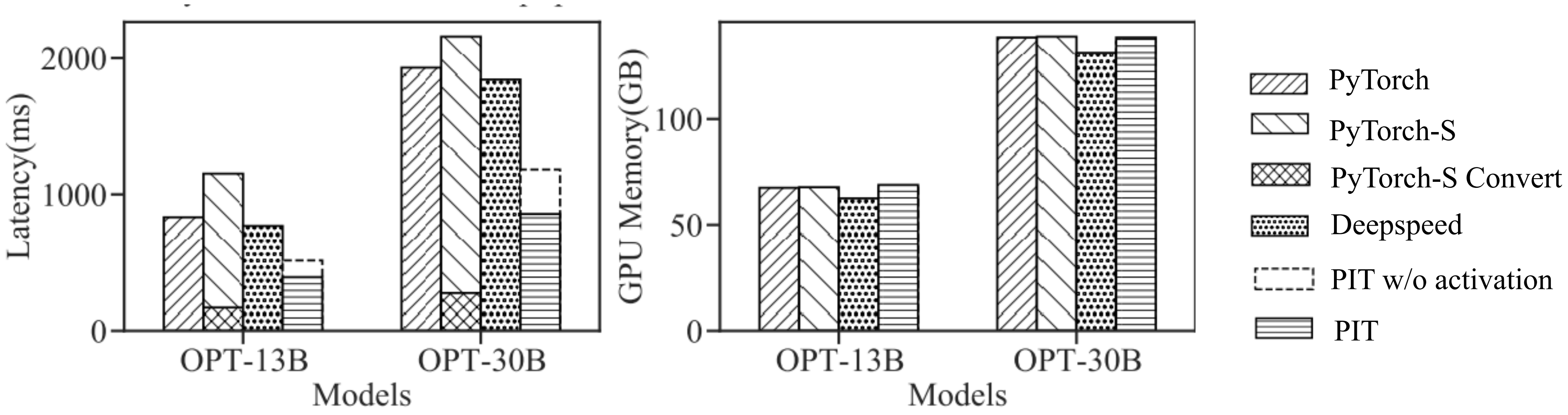
FP 32



Evaluation: End-to-End Inference

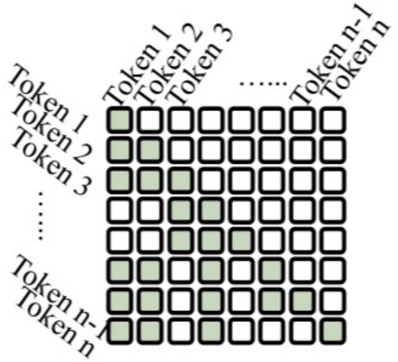
- OPT (8x V100, 13B/30B)
  - Latency: PIT is the lowest
    - Eliminating the padding overhead
    - Exploiting fine-grained sparsity in **ReLU** activation
  - Memory: DeepSpeed is the lowest
    - Deepspeed **fuse** a layer into one operator





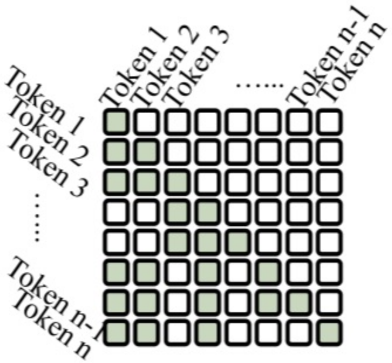
Evaluation: End-to-End Inference

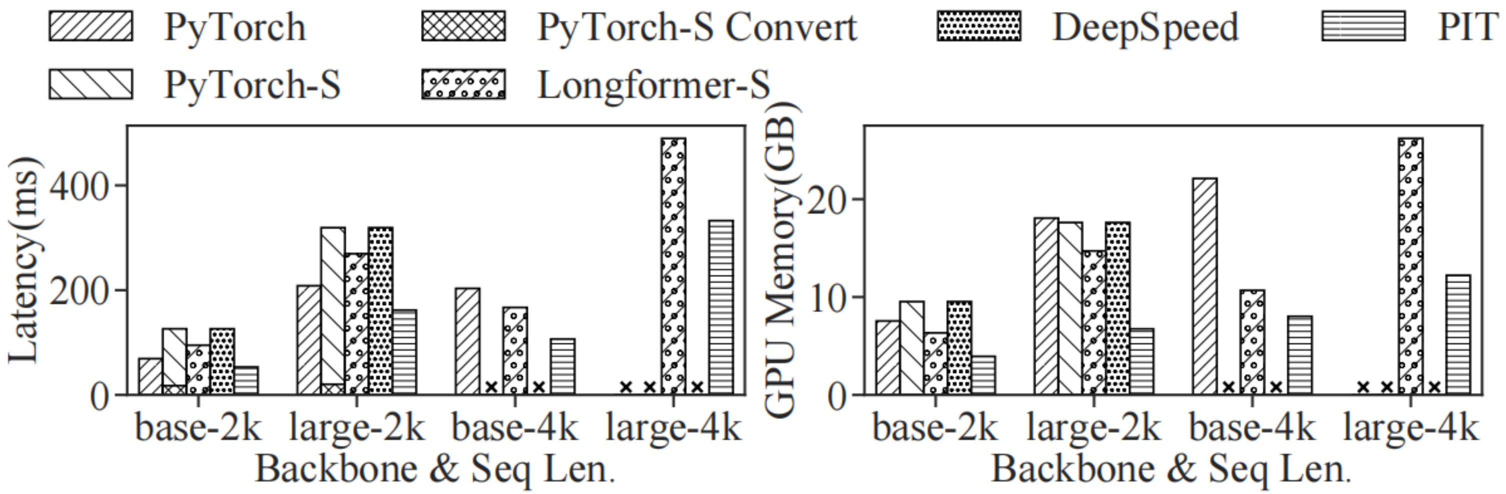
- Longformer(1x V100, FP32) Settings:
  - Sparsity: **Dynamic attention**
  - Input length: 2048/ 4096
  - Baselines:
    - Add **Longformer-S**
      - The sparse implementation specifically optimized for the Longformer
    - PyTorch-S and Deepspeed both selects **Triton** as the backend



Evaluation: End-to-End Inference

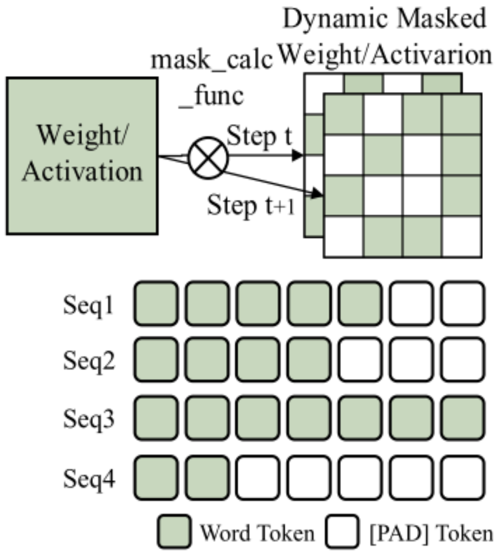
- Longformer(1x V100, FP32)
  - Latency: Longformer-S stands out, but PIT is better
    - Longformer-S: specifically optimized GPU kernels
    - PIT: no large data rearrangement overheads
  - Memory: PIT is the lowest
    - Without data re-arrangement (without **intermediate tensors**)

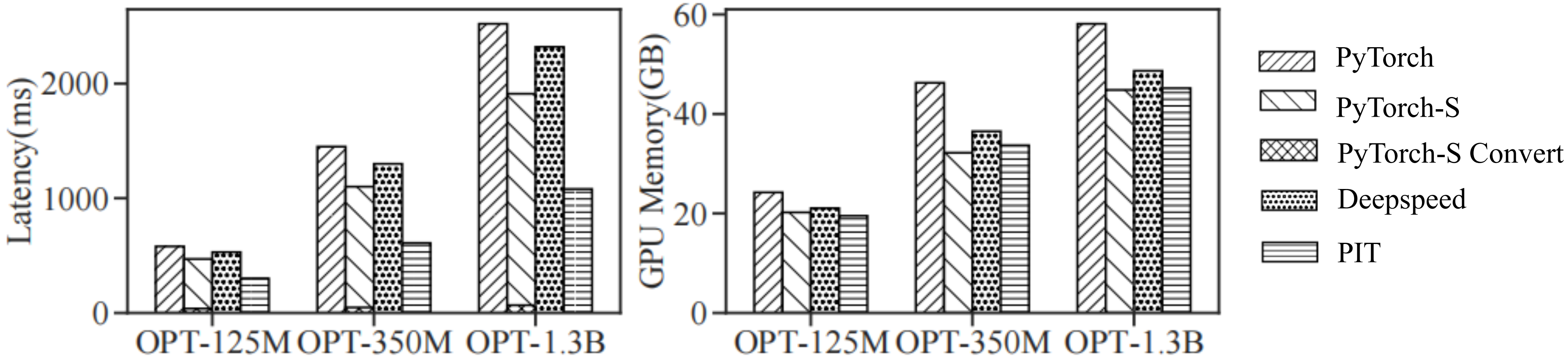




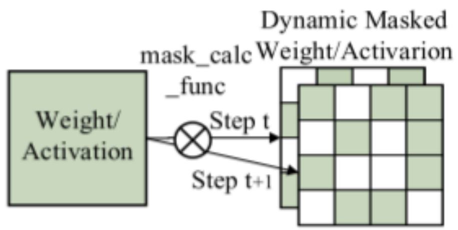
Evaluation: End-to-End Training

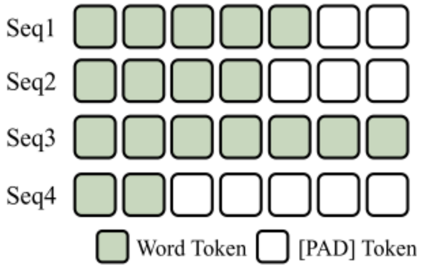
- OPT Training(1x A100, 125M/350M/1.3B)
  - Latency: PIT is the lowest
    - Without padding
    - Supports more fine-grained sparsity granularity
  - Memory: PIT is the lowest
    - Avoid reformatting data from dense to sparse formats





Evaluation: End-to-End Training

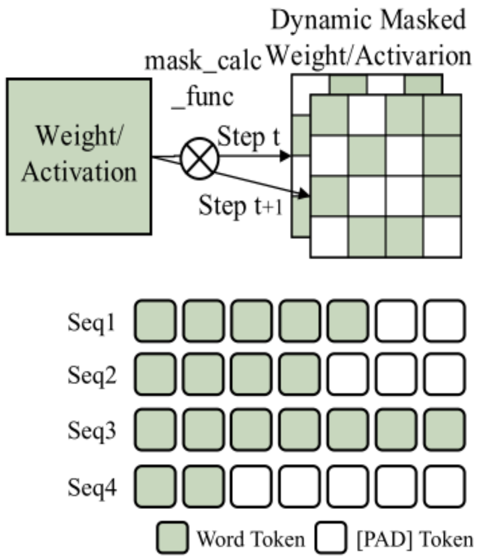


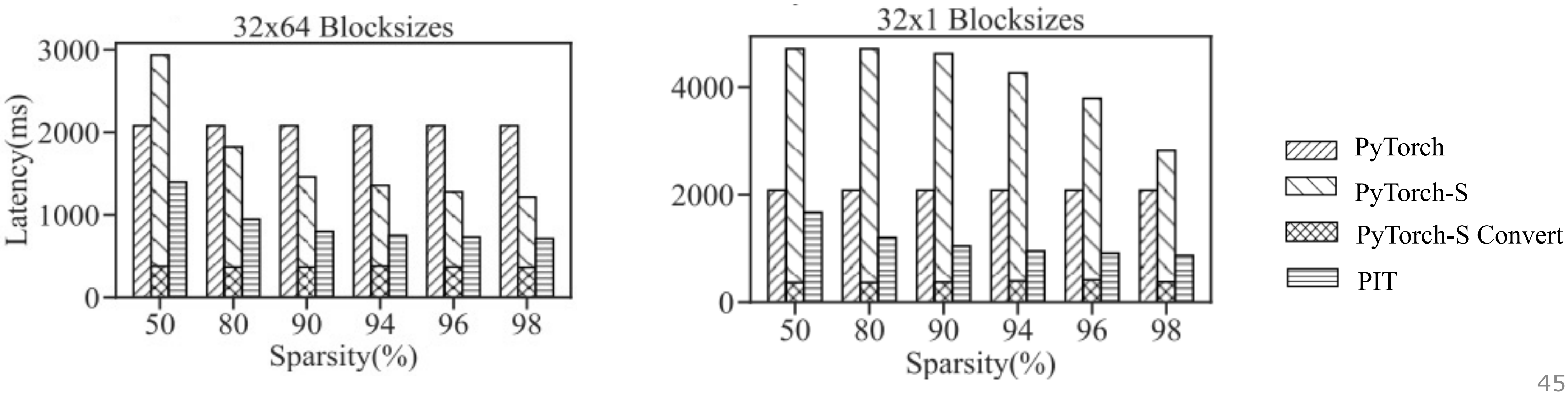


- BERT Training (1x V100) Settings:
  - **Iterative Pruning**
    - Generates a mask based on the weight's magnitude
  - Pruned using block-wise sparsity at **two granularities**: 32×64 and 32×1

Evaluation: End-to-End Training

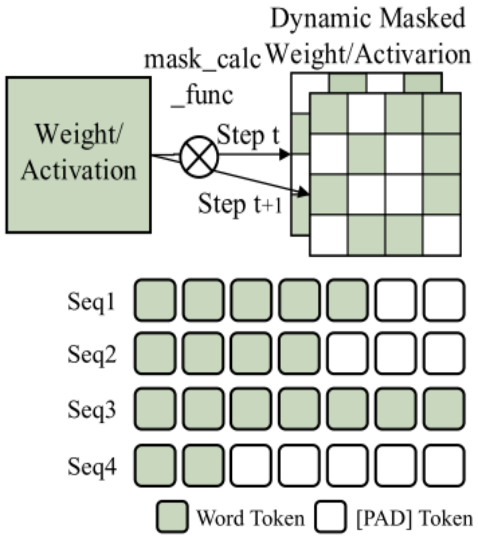
- Granularity: 32×64
  - Latency: PIT is the lowest
    - PyTorch-S suffer from heavy index construction
- Similar trend occurs on granularity of 32×1
  - Performance of PyTorch-S is worse than 32×64
  - But accuracy increases slightly

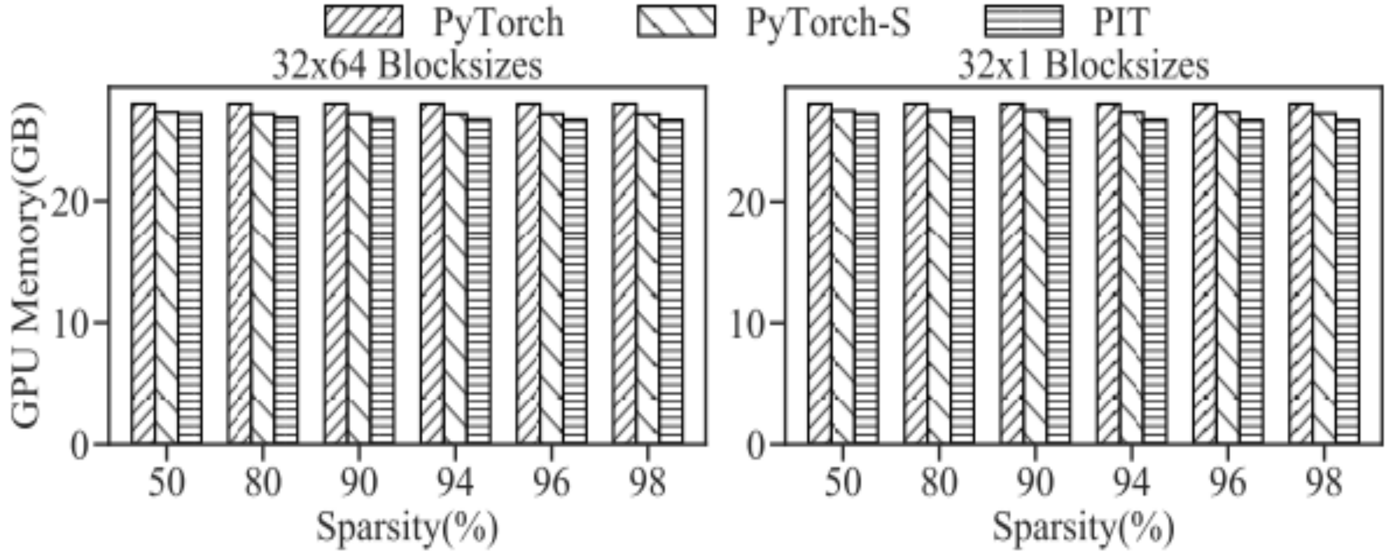




Evaluation: End-to-End Training

- BERT Training (1x V100):
  - Memory: PIT is similar to baselines in both granularity, footprint dropped slightly as sparsity ratio increased
    - Weight tensors take up only a small fraction of memory



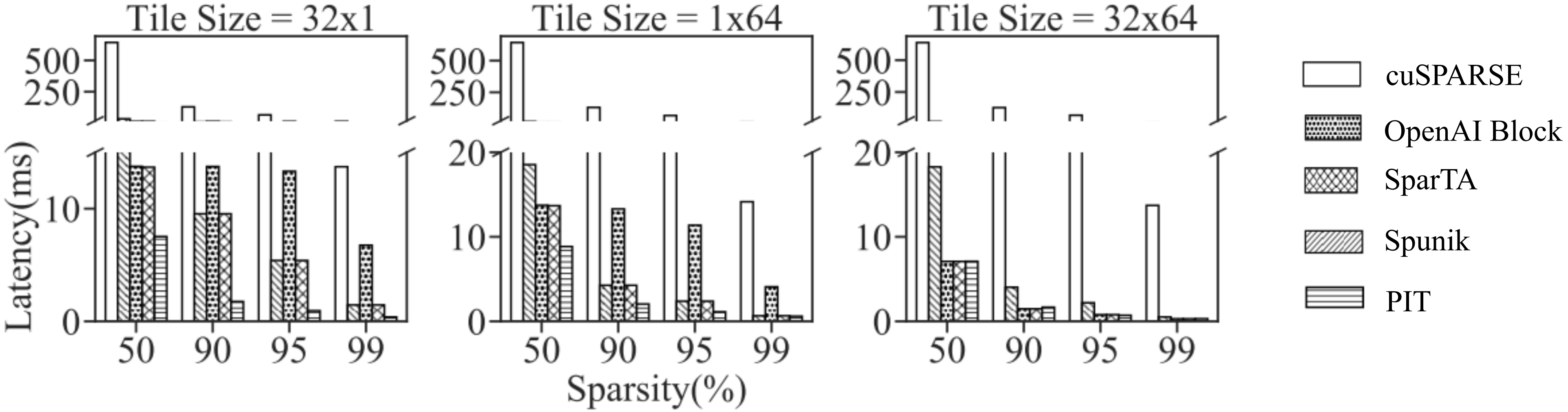


Evaluation: Effectiveness of PIT Transformation

- Exp1: PIT Transformation on Dense Kernels:
  - Experiments Settings:
    - Sparse matrix with different sparsity granularities and shapes
    - Baselines: Sparse libraries, including cuSPARSE, Sputnik, OpenAI Block Sparse (Triton), and **SparTA** (state-of-art **static** sparsity optimization)
    - Use a **static** sparsity pattern to evaluate the computation efficiency

Evaluation: Effectiveness of PIT Transformation

- Exp1: PIT Transformation on Dense Kernels:
  - PIT, SparTA, and OpenAI Block Sparse have **similar latency in 32×64**
    - They use the same dense computation tile
  - cuSPARSE and Sputnik perform poorly
    - High conversion overheads
    - Poor kernels implementations
  - PIT perform **best** in **32×1 and 1×64**
    - support changing smaller micro tiles under small sparsity granularity

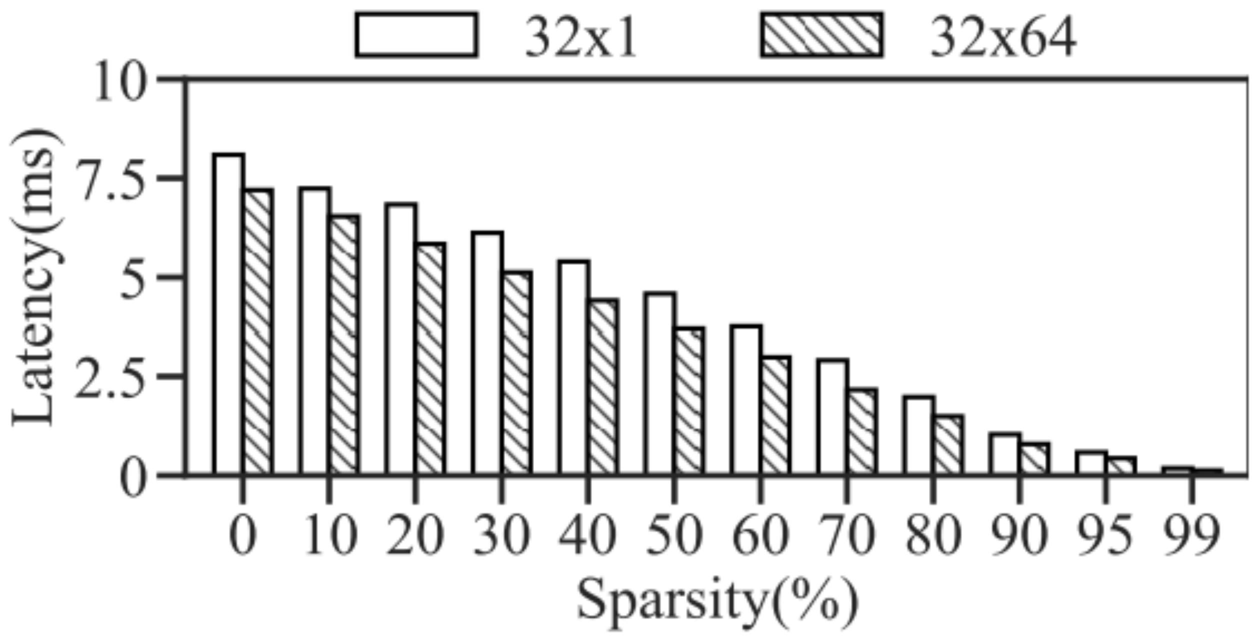


Evaluation: Effectiveness of PIT Transformation

- Exp2: PIT transformation on hardware instructions:
  - Purpose: Show PIT transformation can adapt to the constraints of hardware instructions
  - Experiments Settings:
    - Two different sparsity granularities: 32×1 and 32×64
    - [4096, 4096]×[4096, 4096] matrix multiplication
    - Hardware instructions: **Wmma**, only supports three shapes ([16, 16]×[16, 16],[32, 8]×[8, 16], [8, 32]×[32, 16]) in half-precision

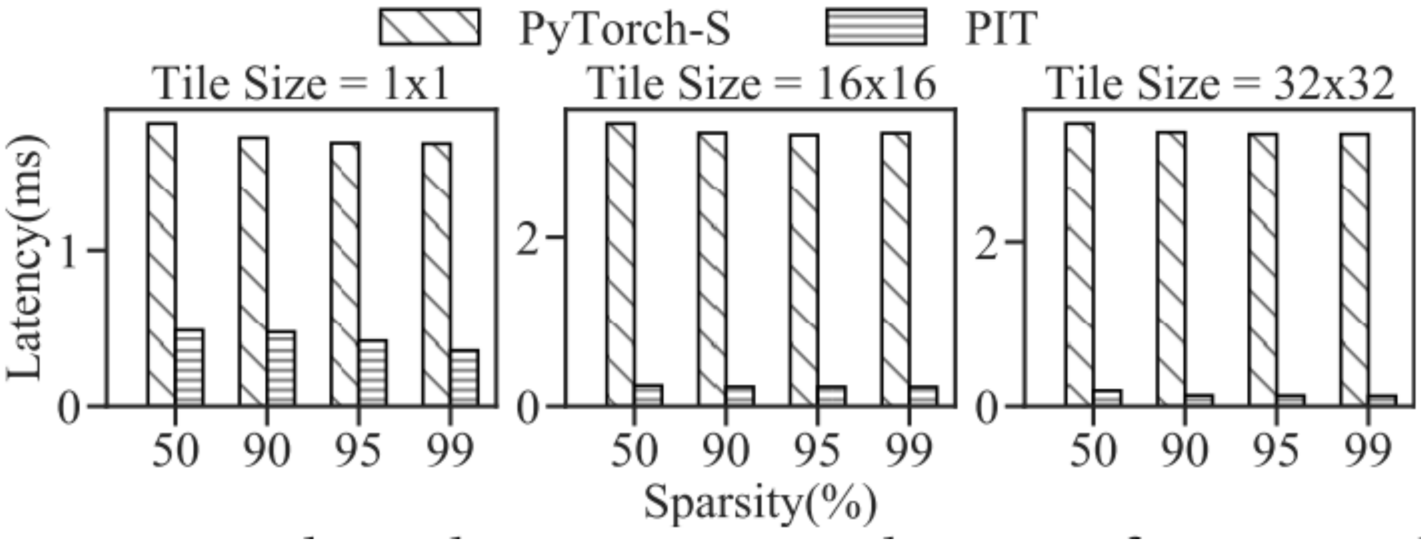
Evaluation: Effectiveness of PIT Transformation

- Exp2: PIT transformation on hardware instructions:
  - The two sparse kernels generated by PIT has **similar latency** at different sparsity ratios
  - PIT transformation introduces little overhead



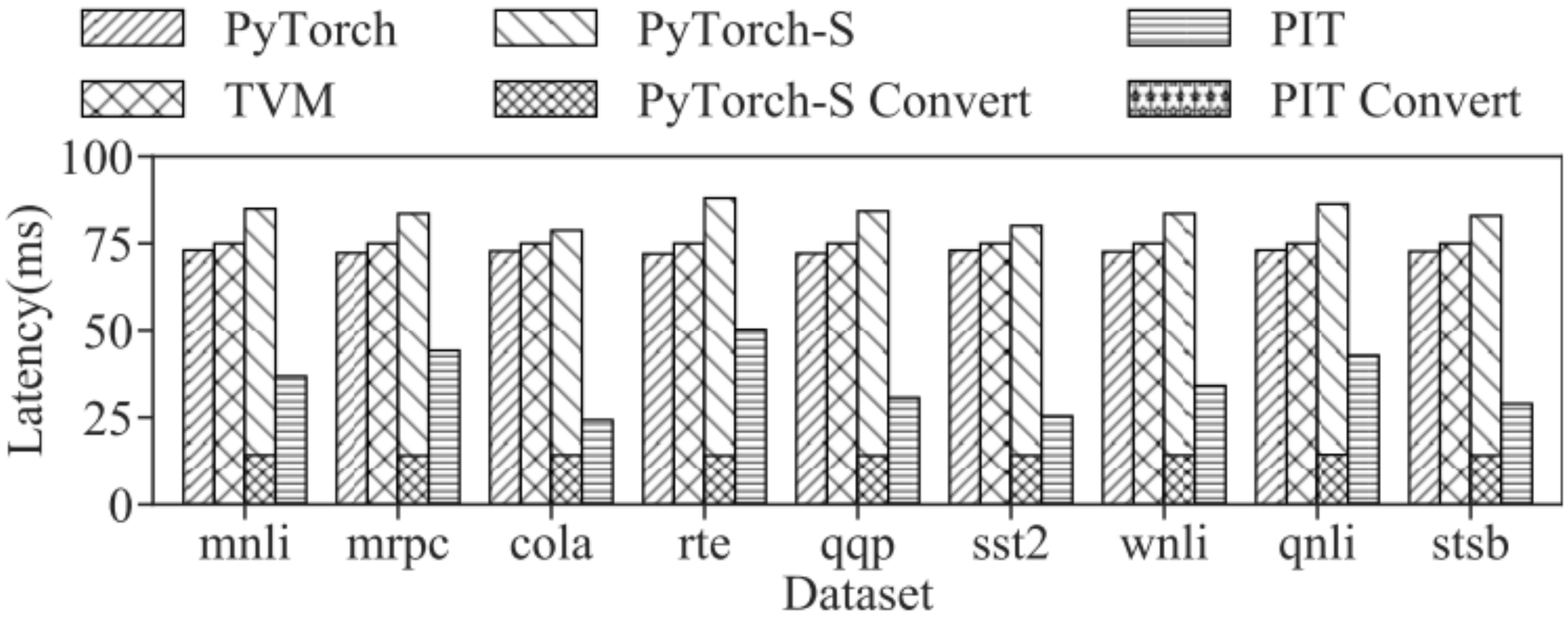
Evaluation: Conversion Overhead

- Exp1: Comparion of conversion latency(1x V100):
  - Settings:
    - Different sparsity granularities and sparsity ratios
  - Convert Latency:
    - PIT is 3.6x~4.7x faster than cuSPARSE at 1× 1 granularity
    - 11.2x~14.2x faster than Triton at 16× 16 granularity
    - 13.3x~26.5x faster than Triton at 32× 32 granularity



Evaluation: Conversion Overhead

- Exp2: The proportion of the conversion overhead:
  - Conversion accounts for 0.7% to 1.1% of the end-to-end latency



Evaluation: Micro-Tile Online Searching

- Different sparsity patterns and different sparsity ratios may lead to different optimal micro-tiles
  - PIT balance between the efficiency and the waste
  - Cost 30us~100us for PIT to search (fast enough)

Sparsity Granularity	Origin Sparsity Ratio(%)	Micro Tile	Sparsity Ratio After Cover (%)	Origin Dense Kernel	Latency (ms)
(2,1)	95	(16, 1)	66.39	$[16, 32] \times [32, 128]$	8.04
(2,1)	99	(8, 1)	96.06	$[8, 32] \times [32, 128]$	2.34
(4,1)	95	(16, 1)	81.45	$[16, 32] \times [32, 128]$	4.29
(4,1)	99	(16, 1)	96.05	$[16, 32] \times [32, 128]$	1.37
(8,1)	95	(8, 1)	95	$[8, 32] \times [32, 128]$	2.34
(8,1)	99	(32, 1)	96.02	$[32, 64] \times [64, 32]$	0.90
(32, 1)	95	(32, 1)	95	$[32, 64] \times [64, 32]$	0.94
(32, 1)	99	(32, 1)	99	$[32, 64] \times [64, 32]$	0.39

Summary

- Pros:
  - ◆ PIT achieves increased efficiency in dynamic sparsity.
  - ◆ PIT supports various models, including those with static sparsity.
  - ◆ PIT minimizes additional overhead by online sparsity detection.

- Further thoughts:
  - ◆ Trade-off between rearrange granularity & efficiency
  - ◆ Support for different operators
  - ◆ Support for App-level sparsity
  - ◆ Profiling is still heavy